Real Time Digital Signal Processing ECE 4703

Lab 2: Implementation of FIR Filters

Group 10: Marcos Delgado (Github ID: marcosdelgadorosa), Brett Mann (Github ID: BrettMann225)

November 6th, 2025

Introduction

Finite impulse response (FIR) filters are commonly used for their ability to isolate passband frequencies in real time. Digital FIR filters are constructed by delaying input samples and multiplying them with specific coefficients. In this lab we explore FIR filters and their design while examining various forms and analyzing optimization algorithms. We will design a lowpass filter that will be able to differentiate between two DTMF frequencies.

Methods

This lab utilized a provided DSP kit consisting of a RPI4 board and a shielded PCB with a high-end DAC and ADC. An Analog Discovery 2 USB scope was used to monitor signals. The appropriate software was installed on all the hardware. An SSH connection was used to access the RPI4 and run necessary programs such as pipewire. Additionally our filter parameters were calculated by MATLAB's filter designer.

Results

Before we can begin designing the lowpass filter, we must verify the capabilities of our hardware. We must verify the maximum amplitude that the ADC can record. Figure 1, shows the process used to achieve this. It functions by keeping track of the minimum and maximum values from the input over all frames. The user can then increase the amplitude of the wave generator until finding where the maximum and minimum inputs stay stagnant.

```c
int process (jack_nframes_t nframes, void *arg) {
  jack_default_audio_sample_t *in, *out;

  int flagl = 0;
  int flagr = 0;

  in = jack_port_get_buffer (input_port_left, nframes);
  out = jack_port_get_buffer (output_port_left, nframes);

  for (int i=0; i < nframes; i++) {
        if (leftmax < in[i]) {
                leftmax = in[i];
                flagl = 1;
                }
        if(leftmin > in[i]) {
                leftmin = in[i];
                flagl = 1;
                }
        }

  memcpy (out, in,
          sizeof (jack_default_audio_sample_t) * nframes);

  in = jack_port_get_buffer (input_port_right, nframes);
  out = jack_port_get_buffer (output_port_right, nframes);

 for (int i=0; i < nframes; i++) {
        if (rightmax < in[i]) {
                rightmax = in[i];
                flagr = 1;
                }
        if(rightmin > in[i]) {
                rightmin = in[i];
                flagr = 1;
                }
        }

  memcpy (out, in,
          sizeof (jack_default_audio_sample_t) * nframes);

  if(flagl) printf("LEFT MIN: %f\nLEFT MAX: %f\n", leftmin, leftmax);
  if(flagr) printf("RIGHT MIN: %f\nRIGHT MAX: %f\n", rightmin, rightmax);


  return 0;
}
```

Figure 1. Process code for verifying ADC capabilities

```
RIGHT MIN: -1.000000
RIGHT MAX: 0.998213
RIGHT MIN: -1.000000
RIGHT MAX: 0.998218
RIGHT MIN: -1.000000
RIGHT MAX: 0.998227
LEFT MIN: -1.000000
LEFT MAX: 0.999931
RIGHT MIN: -1.000000
RIGHT MAX: 0.998442
LEFT MIN: -1.000000
LEFT MAX: 0.999991
RIGHT MIN: -1.000000
RIGHT MAX: 0.998643
LEFT MIN: -1.000000
LEFT MAX: 1.000000
RIGHT MIN: -1.000000
RIGHT MAX: 1.000000
```

Figure 2. Output of the ADC function keeping track of maximum and minimum.

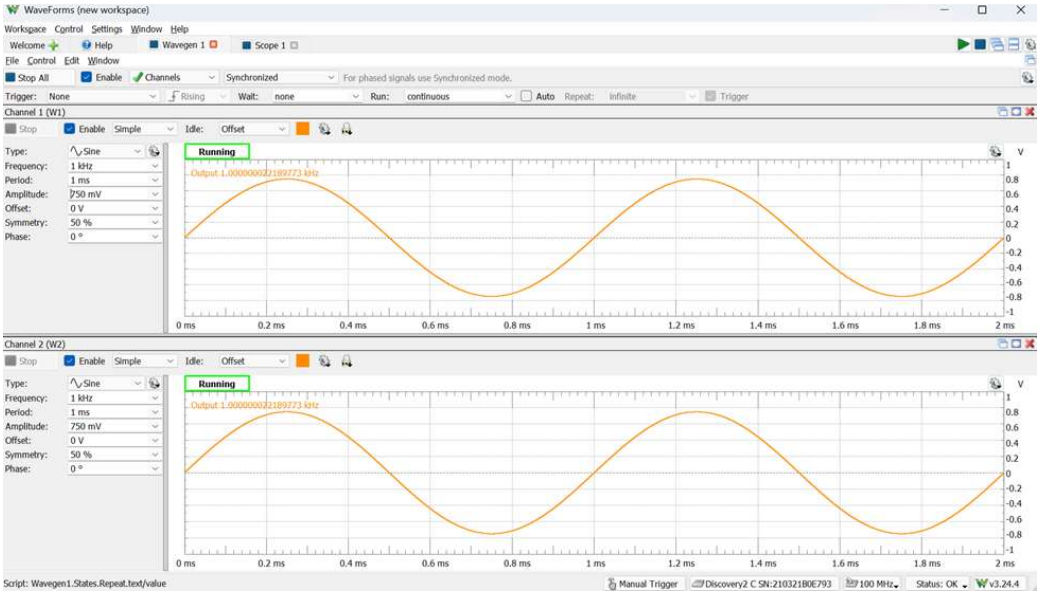


Figure 3. Maximum amplitude of wavegenerators before information loss.

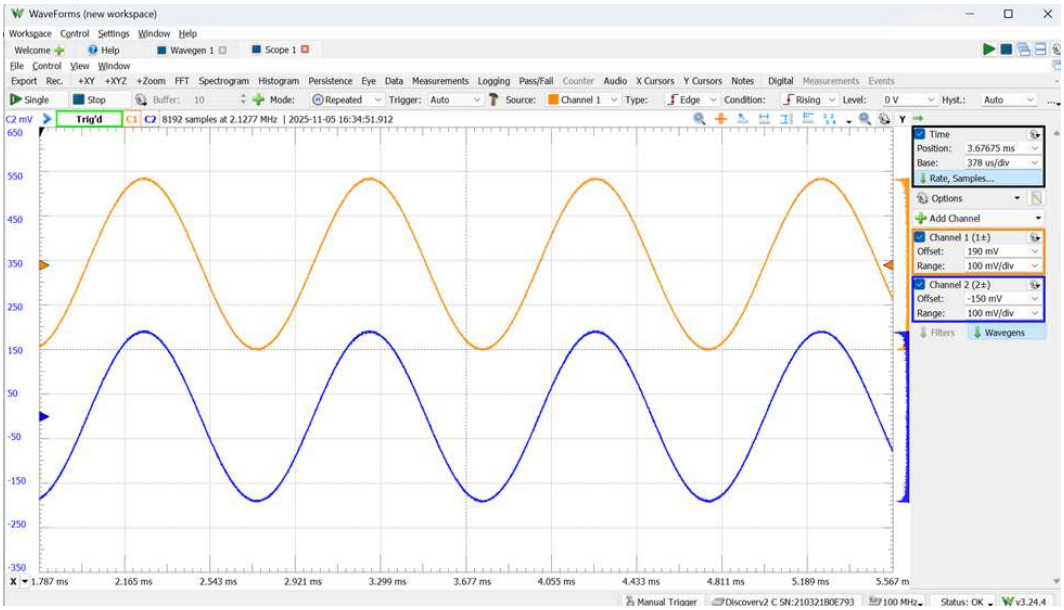


Figure 4. Scope depicting largest possible clean sinewaves, achieved with 750mV input.

Figure 2 demonstrates the output of the program as a user gradually increases the input amplitude. Comparing the plateau to the point at which it was reached, we can determine that Vmax and Vmin are 750mV and -750mV respectively, as demonstrated in figure 3. Noticeably, the minimums and maximums recorded by the system are different, telling us that the values Nmax and Nmin (identical for left and right channel) are 1 and -1 respectively.

```c
float amplitude = 1;
int indexleft = 0;
float max = -1.0f;
float min = 1.0f;
int flag = 0;

int process (jack_nframes_t nframes, void *arg) {
  jack_default_audio_sample_t *outleft, *outright;

  int i;
  outleft = jack_port_get_buffer (output_port_left, nframes);

  float phase;
  for (i=0; i<nframes; i++) {
        phase = (indexleft)/47.0f;
        outleft[i] = phase*amplitude*2-amplitude;
        indexleft = (indexleft+1)%48;
        if (outleft[i] < min){
                min = outleft[i];
                flag = 1;
                }
        if (outleft[i] > max){
                max = outleft[i];
                flag = 1;
        }
  }


amplitude += 0.01;

  if (flag == 1) {
        printf("Amplitude: %f\nMin: %f\nMax: %f\n", amplitude, min, max);
  }

flag =0;
  return 0;

}

void jack_shutdown (void *arg) {
        exit (1);
}
```

Figure 5. Code used to determine minimum and maximum values of the DAC.

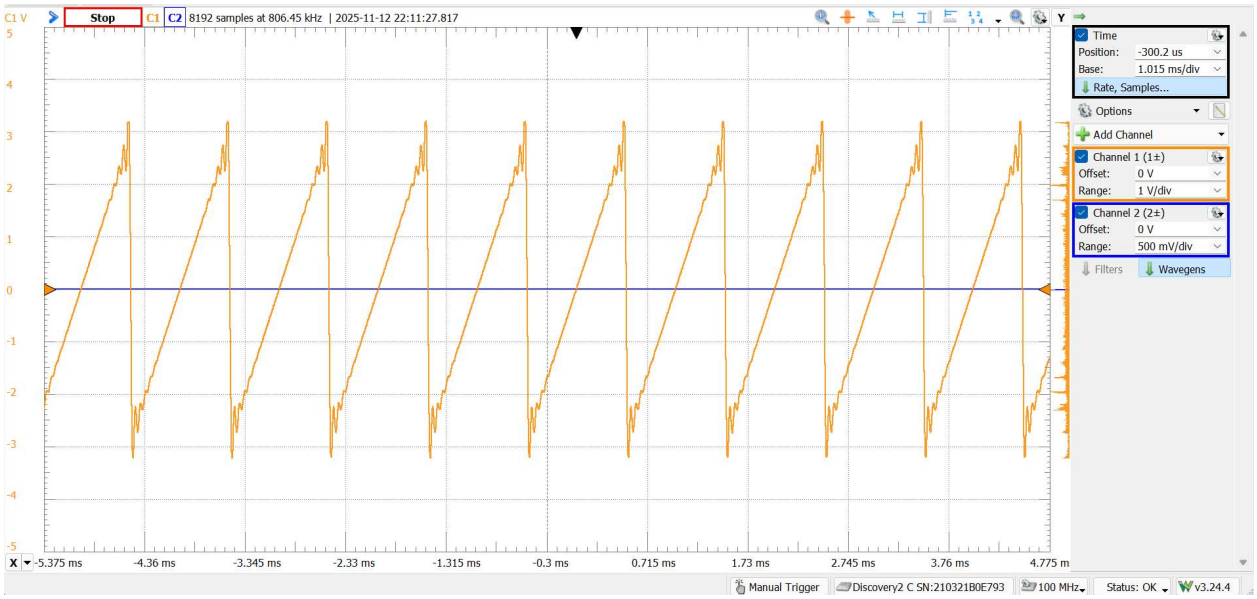


Figure 6. Triangle wave depicting the maximum amplitude of about 3.2V

To verify the characteristics of the DAC, we must generate the wave that we send out. Figure 5 shows the program used to generate a sawtooth that gradually increases in amplitude. The program works by linearly mapping each sample in the cycle to a point from the negative amplitude to the maximum and then repeating at a rate of 1kHz. Using this the user can watch the scope and view where it begins to clip the top. Figure 6 shows the sawtooth right as the clipping appears, depicting Vmax and Vmin for the DAC to be about 3.2V and -3.2V respectively. This corresponded to an amplitude of about 15 meaning that Ndac is equal to 15.

It is now possible to effectively design a filter for our given parameters. We want to be able to separate the left two columns of DTMF frequencies by filtering out the right two. In order to do this we need a lowpass filter with a cutoff of 1336 Hz and a stopband starting at 1477 Hz, a maximum passband ripple of 0.2dB and stopband rejection of 50 dB. We use MATLAB filter designer to get an equiripple filter and it gives us a 750 tap filter. We can see the rejection in figure 7, showing the cutoff at 1336 Hz. In figure 8 we can see the impulse response which is a ripple with a sharp spike in the middle. Figure 9 depicts the phase response and we can verify that our filter is indeed linear as the plot follows linear relationships. The pole zero plot in figure 10 depicts all 750 zeros which create a circle around 0.

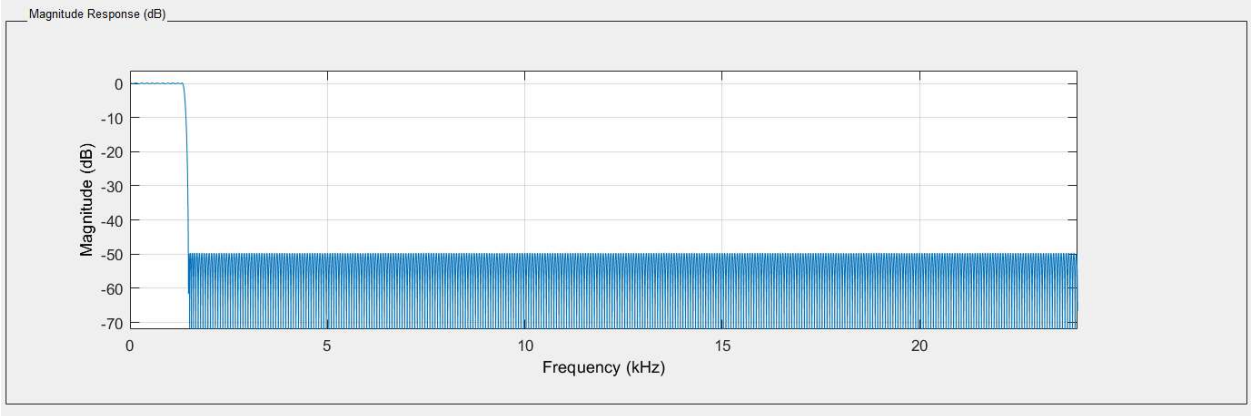


Figure 7. Magnitude response of filter.

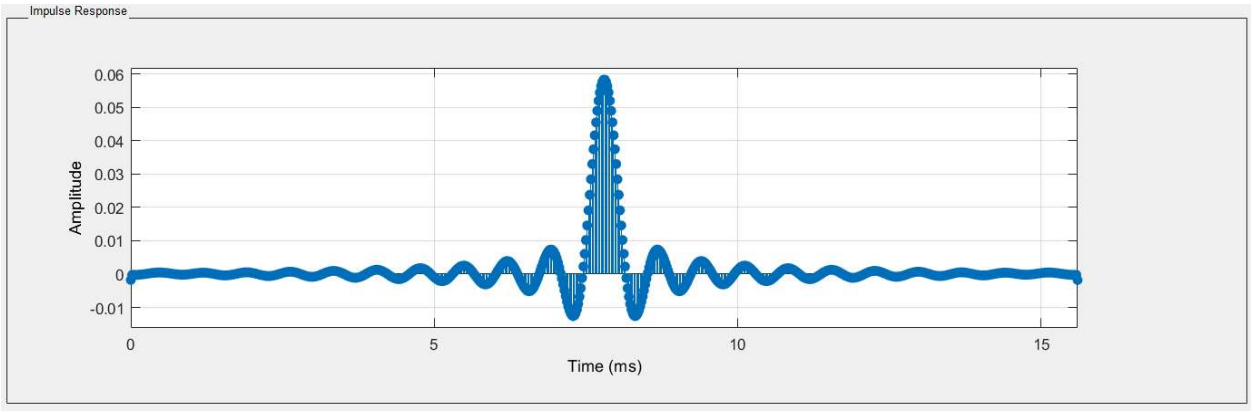


Figure 8. Impulse response of filter.

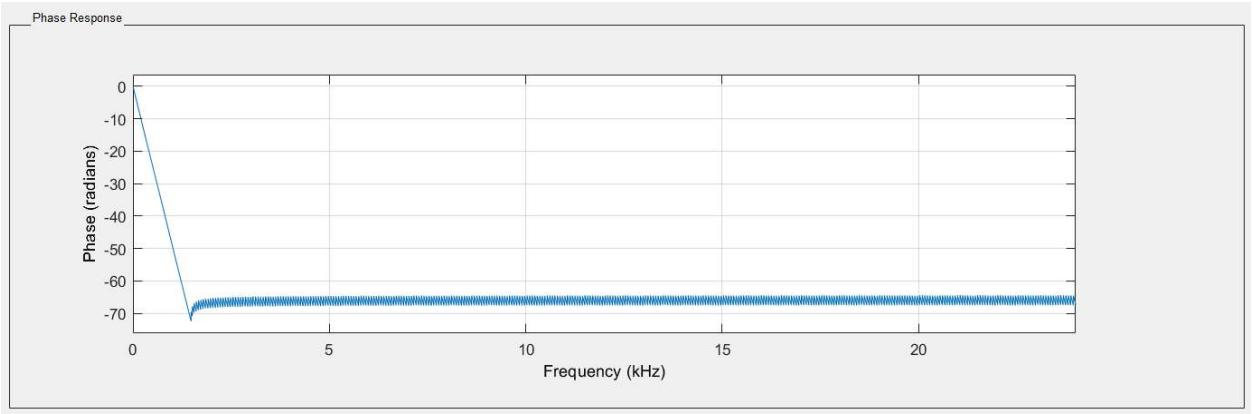


Figure 9. Phase response of filter.

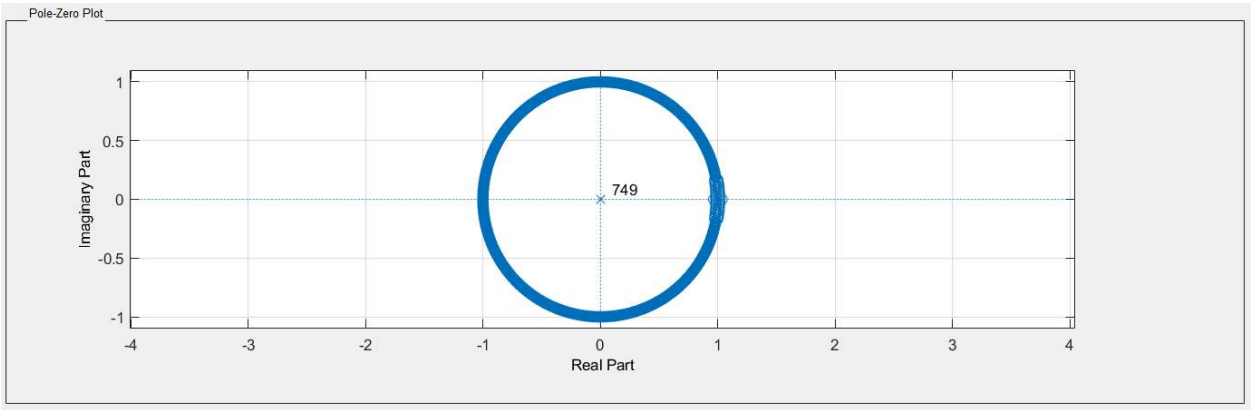
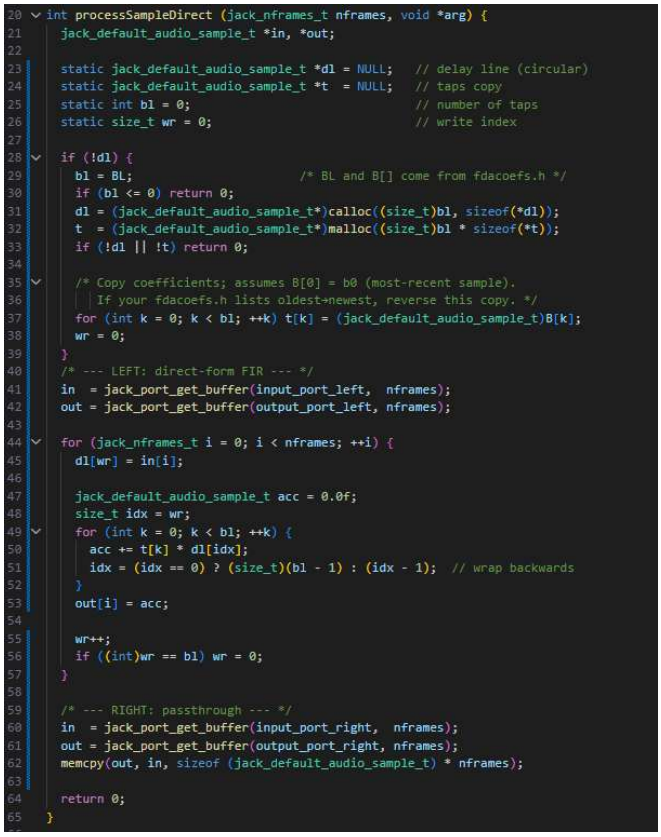


Figure 10. Pole-zero plot of filter.

We use the coefficients found by MATLAB to implement our desired lowpass filter. We will use a direct form FIR filter meaning that we will store a delayed signal and then calculate the multiplication with the coefficients. As shown in Figure 11, we define a circular delay line which will continuously keep track of the last 750 inputs (initialized to 0). For every input we then

multiply each value in the delay line with the corresponding delay coefficient (imported from MATLAB).



```c
int processSampleDirect (jack_nframes_t nframes, void *arg) {
  jack_default_audio_sample_t *in, *out;

  static jack_default_audio_sample_t *dl = NULL;   // delay line (circular)
  static jack_default_audio_sample_t *t  = NULL;   // taps copy
  static int bl = 0;                               // number of taps
  static size_t wr = 0;                            // write index

  if (!dl) {
    bl = BL;                          /* BL and B[] come from fdacoefs.h */
    if (bl <= 0) return 0;
    dl = (jack_default_audio_sample_t*)calloc((size_t)bl, sizeof(*dl));
    t  = (jack_default_audio_sample_t*)malloc((size_t)bl * sizeof(*t));
    if (!dl || !t) return 0;

    /* Copy coefficients; assumes B[0] = b0 (most-recent sample).
       If your fdacoefs.h lists oldest→newest, reverse this copy. */
    for (int k = 0; k < bl; ++k) t[k] = (jack_default_audio_sample_t)B[k];
    wr = 0;
  }
  /* --- LEFT: direct-form FIR --- */
  in  = jack_port_get_buffer(input_port_left,  nframes);
  out = jack_port_get_buffer(output_port_left, nframes);

  for (jack_nframes_t i = 0; i < nframes; ++i) {
    dl[wr] = in[i];

    jack_default_audio_sample_t acc = 0.0f;
    size_t idx = wr;
    for (int k = 0; k < bl; ++k) {
      acc += t[k] * dl[idx];
      idx = (idx == 0) ? (size_t)(bl - 1) : (idx - 1);  // wrap backwards
    }
    out[i] = acc;

    wr++;
    if ((int)wr == bl) wr = 0;
  }

  /* --- RIGHT: passthrough --- */
  in  = jack_port_get_buffer(input_port_right,  nframes);
  out = jack_port_get_buffer(output_port_right, nframes);
  memcpy(out, in, sizeof (jack_default_audio_sample_t) * nframes);

  return 0;
}
```

Figure 11. Process for direct form FIR filter.

Noticing that the process only filters the left channel, we can use the right channel to compare the unfiltered signal. Figure 12 shows how for the 445 Hz square wave we are only able to see a 2 sine wave combination, which makes sense as the square wave is composed of odd harmonics and only 445Hz and 1335 Hz (3 * 445Hz) falls within the passband. However, the 492 Hz square wave has its first harmonic at 1476 Hz, well outside the passband, resulting in the single sine wave depicted in Figure 13. Additionally, we can verify that our solution meets the processor load threshold by observing that the errors are at 0 as depicted in figure 14.

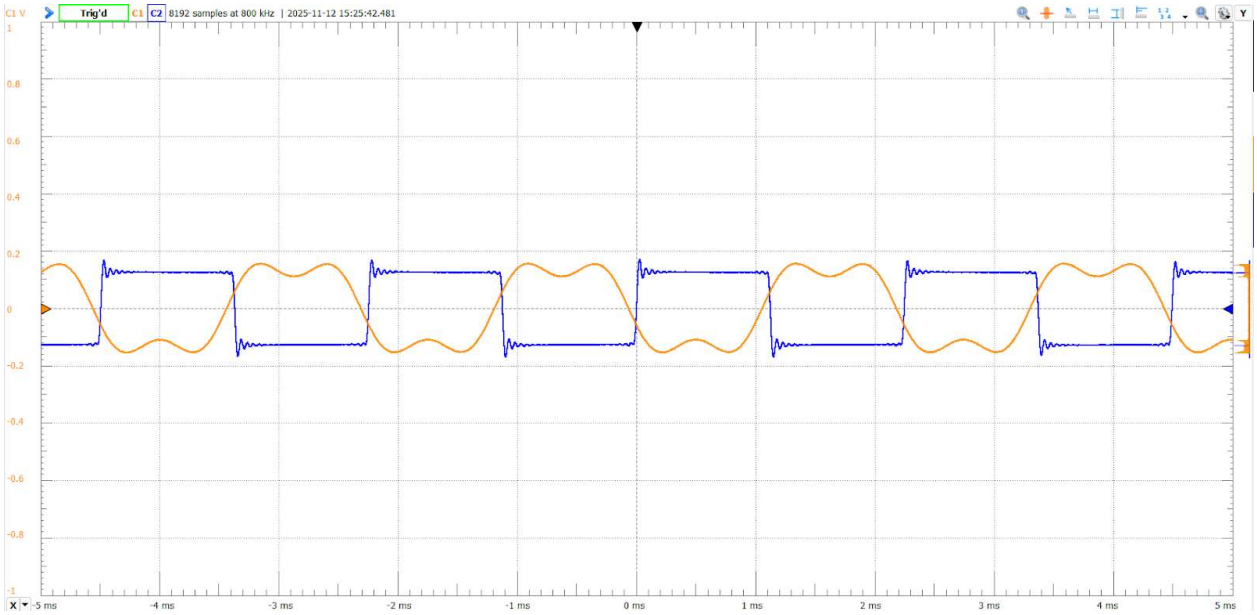


Figure 12. 445Hz square waves. Left (orange) is passed through the filter, right (blue) is not.

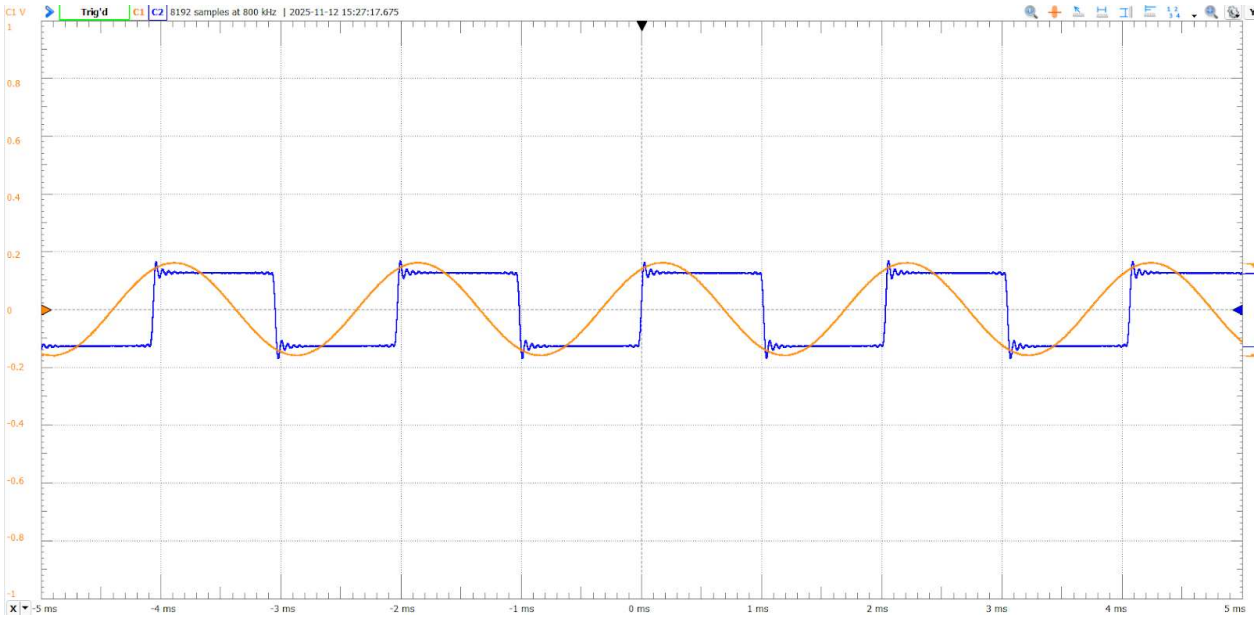


Figure 13. 492Hz square waves. Left (orange) is passed through the filter, right (blue) is not.

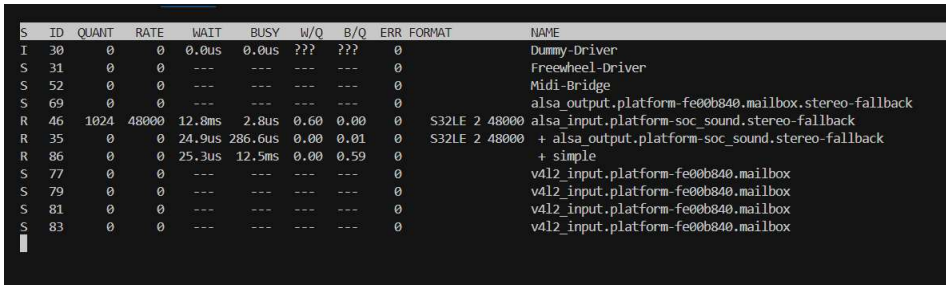
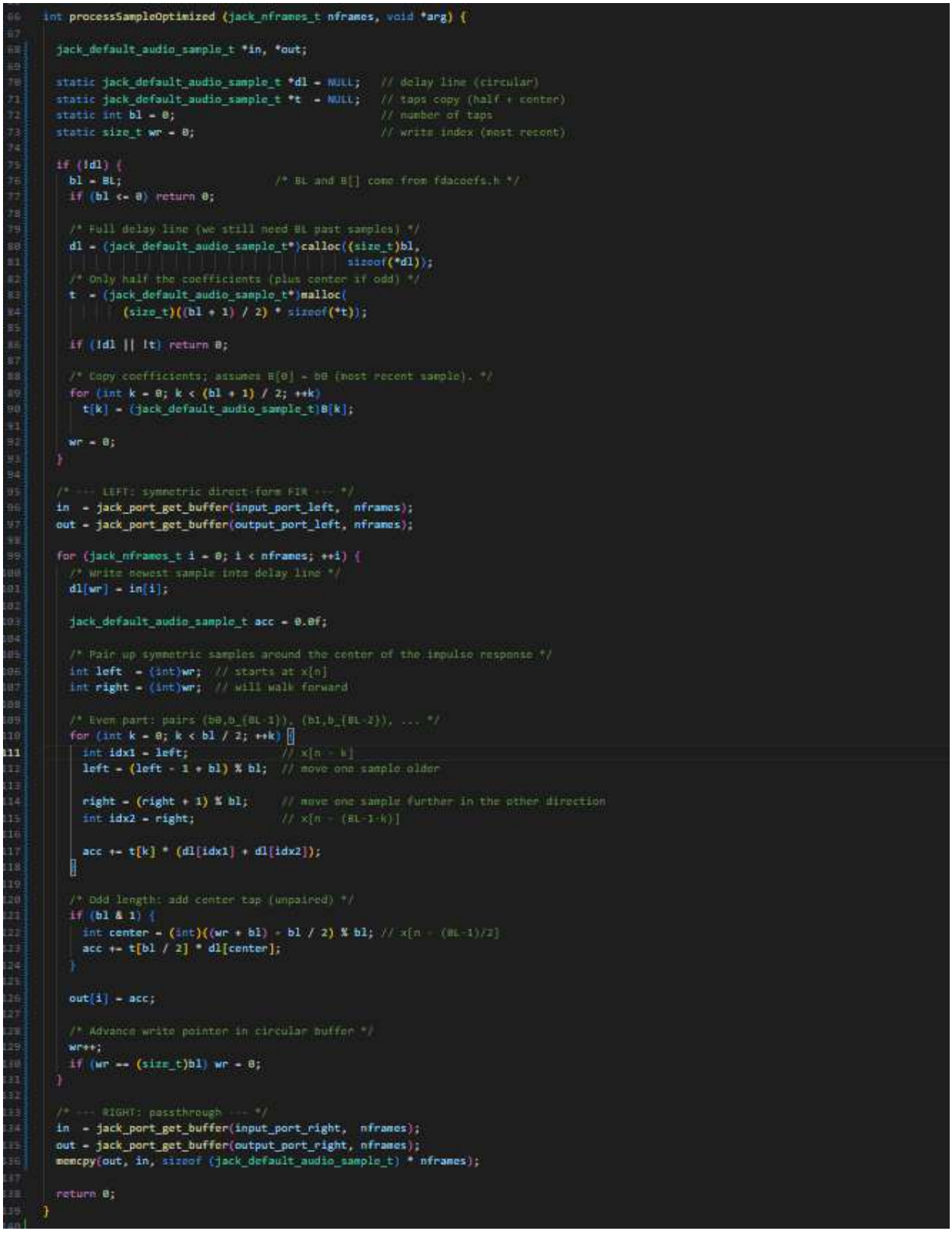


Figure 14. Direct form Filter performance.

Given that we have designed a linear filter, there is a simple optimization that is possible. The coefficients are all reflected therefore we can afford to store an array of half the length. Figure 15 depicts the process which now uses a coefficient array, t, of half the length and uses clever step tracking to link up opposite ends of the list, as they share coefficients. The loop then steps in both directions to calculate all pairs together for the entire delay line. Figure 16 shows the filtering of the 445 Hz square, identical to before and similarly figure 17 depicts the 492 Hz under the optimized program. Figure 18 reveals that our optimized program continues to have 0 errors and also operates significantly faster.



```c
int processSampleOptimized (jack_nframes_t nframes, void *arg) {

  jack_default_audio_sample_t *in, *out;

  static jack_default_audio_sample_t *dl = NULL;   // delay line (circular)
  static jack_default_audio_sample_t *t  = NULL;   // taps copy (half + center)
  static int bl = 0;                               // number of taps
  static size_t wr = 0;                            // write index (most recent)

  if (!dl) {
    bl = BL;                      /* BL and B[] come from fdacoefs.h */
    if (bl <= 0) return 0;

    /* Full delay line (we still need BL past samples) */
    dl = (jack_default_audio_sample_t*)calloc((size_t)bl,
                                      sizeof(*dl));
    /* Only half the coefficients (plus center if odd) */
    t  = (jack_default_audio_sample_t*)malloc(
          (size_t)((bl + 1) / 2) * sizeof(*t));

    if (!dl || !t) return 0;

    /* Copy coefficients; assumes B[0] = b0 (most recent sample). */
    for (int k = 0; k < (bl + 1) / 2; ++k)
      t[k] = (jack_default_audio_sample_t)B[k];

    wr = 0;
  }

  /* --- LEFT: symmetric direct-form FIR --- */
  in  = jack_port_get_buffer(input_port_left,  nframes);
  out = jack_port_get_buffer(output_port_left, nframes);

  for (jack_nframes_t i = 0; i < nframes; ++i) {
    /* Write newest sample into delay line */
    dl[wr] = in[i];

    jack_default_audio_sample_t acc = 0.0f;

    /* Pair up symmetric samples around the center of the impulse response */
    int left  = (int)wr;  // starts at x[n]
    int right = (int)wr;  // will walk forward

    /* Even part: pairs (b0,b_{BL-1}), (b1,b_{BL-2}), ... */
    for (int k = 0; k < bl / 2; ++k) {
      int idx1 = left;              // x[n - k]
      left = (left - 1 + bl) % bl;  // move one sample older

      right = (right + 1) % bl;     // move one sample further in the other direction
      int idx2 = right;             // x[n - (BL-1-k)]

      acc += t[k] * (dl[idx1] + dl[idx2]);
    }

    /* Odd length: add center tap (unpaired) */
    if (bl & 1) {
      int center = (int)((wr + bl) - bl / 2) % bl; // x[n - (BL-1)/2]
      acc += t[bl / 2] * dl[center];
    }

    out[i] = acc;

    /* Advance write pointer in circular buffer */
    wr++;
    if (wr == (size_t)bl) wr = 0;
  }

  /* --- RIGHT: passthrough --- */
  in  = jack_port_get_buffer(input_port_right,  nframes);
  out = jack_port_get_buffer(output_port_right, nframes);
  memcpy(out, in, sizeof (jack_default_audio_sample_t) * nframes);

  return 0;
}
```

Figure 15. The optimized code FIR depicting t of half the length and the calculation of symmetric samples by stepping left and right around the circular array.

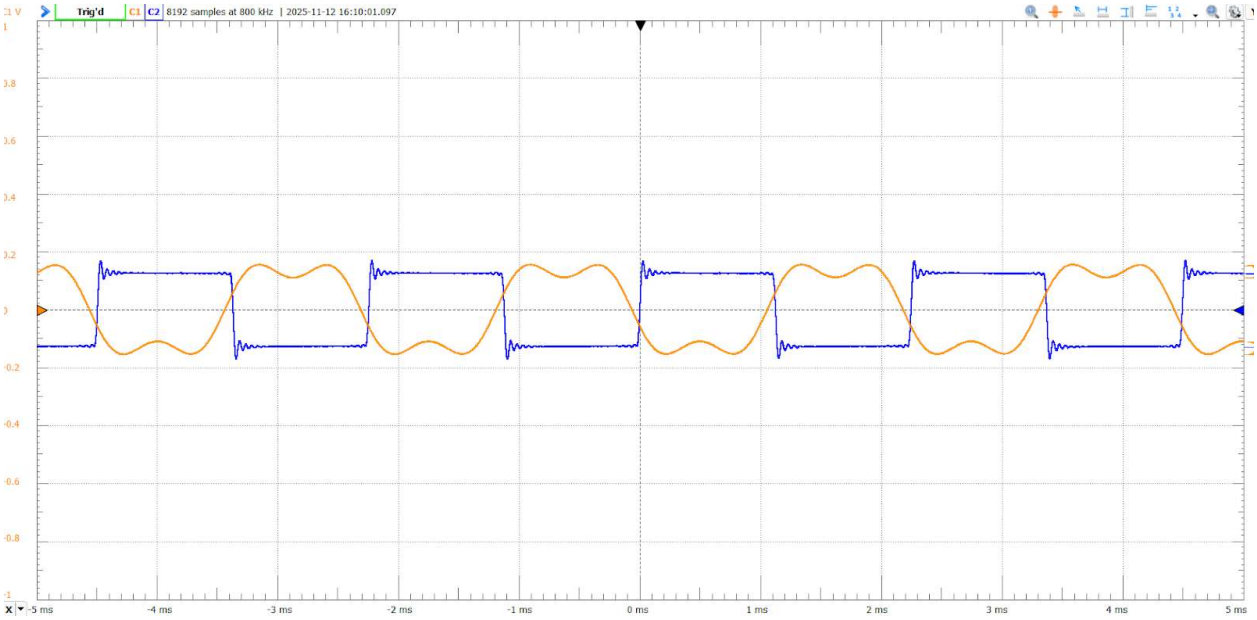


Figure 16. 445 Hz square waves through the optimized filter. Left (orange) is passed through the filter, right (blue) is not

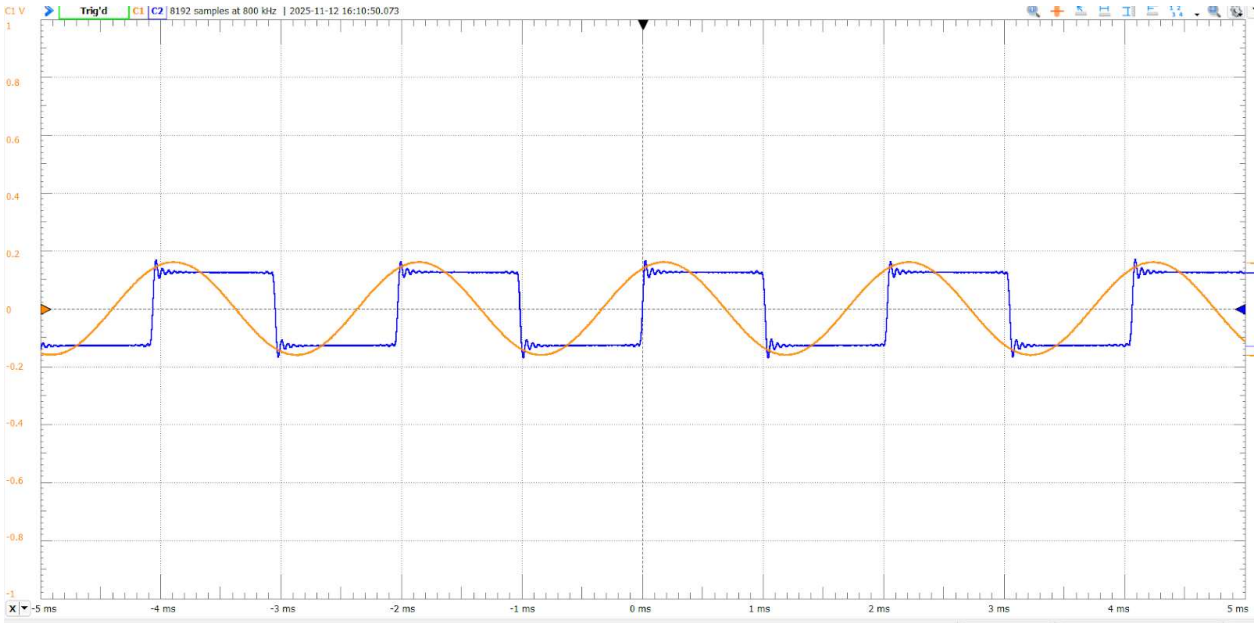


Figure 17. 492 Hz square waves through the optimized filter. Left (orange) is passed through the filter, right (blue) is not

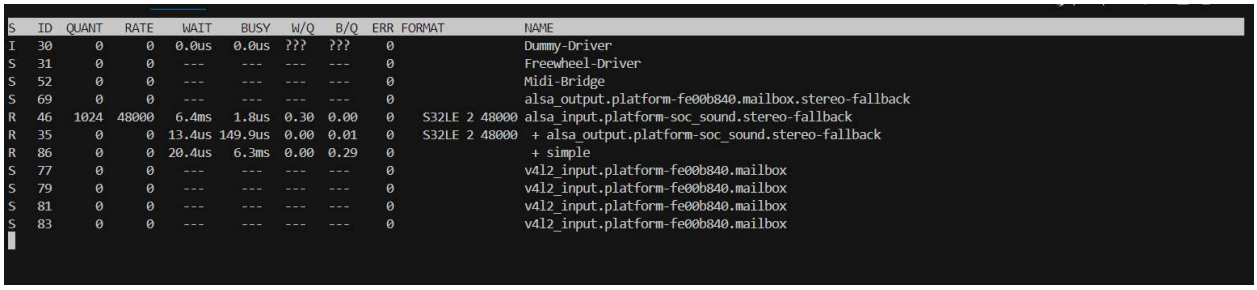


Figure 18. Performance of optimized direct form FIR filter

Instead of the direct form, it is possible to use the transpose form, where intermediate values are stored in the taps rather than a delay line. To do this we create an array to store 750 values where each is the current sample multiplied by the coefficient that corresponds to that delay, as seen in Figure 19. For every sample, we set the first entry to the addition of the previous second entry and the current sample multiplied by the 0th coefficient. Each following entry is similarly set to the addition of the entry preceding it and the multiplication of the current sample and a coefficient. This process will continuously update the array and give a result that is identical to the direct form FIR. Figure 20 appears identical to that of Figure 12 which is expected as though they are coded differently, the filter is the same. Figure 21 similarly is identical to the graph for the direct form case in figure 13. The performance shown in figure 22 seems to be improved over that of the direct form from earlier implying that this configuration is easier computationally.

```c
int processSampleTransposed (jack_nframes_t nframes, void *arg) {

  jack_default_audio_sample_t *in, *out;

  static jack_default_audio_sample_t *t  = NULL;   // taps copy
  static jack_default_audio_sample_t *s  = NULL;   // transposed-form states
  static int bl = 0;                               // number of taps

  if (!t) {
    bl = BL;                      /* BL and B[] come from fdacoefs.h */
    if (bl <= 0) return 0;

    /* Copy coefficients */
    t = (jack_default_audio_sample_t*)malloc((size_t)bl * sizeof(*t));
    if (!t) return 0;
    for (int k = 0; k < bl; ++k)
      t[k] = (jack_default_audio_sample_t)B[k];

    /* Allocate states: BL-1 of them (0 if BL == 1) */
    if (bl > 1) {
      s = (jack_default_audio_sample_t*)calloc((size_t)(bl - 1),
                                               sizeof(*s));
      if (!s) return 0;
    }
  }

  /* --- LEFT: transposed-form FIR --- */
  in  = jack_port_get_buffer(input_port_left,  nframes);
  out = jack_port_get_buffer(output_port_left, nframes);

  for (jack_nframes_t i = 0; i < nframes; ++i) {
    jack_default_audio_sample_t x = in[i];
    jack_default_audio_sample_t y;

    if (bl == 1) {

      y = t[0] * x;
    } else {

      y = t[0] * x + s[0];


      for (int k = 0; k < bl - 2; ++k) {
        s[k] = t[k + 1] * x + s[k + 1];
      }
      s[bl - 2] = t[bl - 1] * x;
    }

    out[i] = y;
  }

  /* --- RIGHT: passthrough --- */
  in  = jack_port_get_buffer(input_port_right,  nframes);
  out = jack_port_get_buffer(output_port_right, nframes);
  memcpy(out, in, sizeof (jack_default_audio_sample_t) * nframes);


  return 0;
}
```

Figure 19. Code for the transposed form FIR.

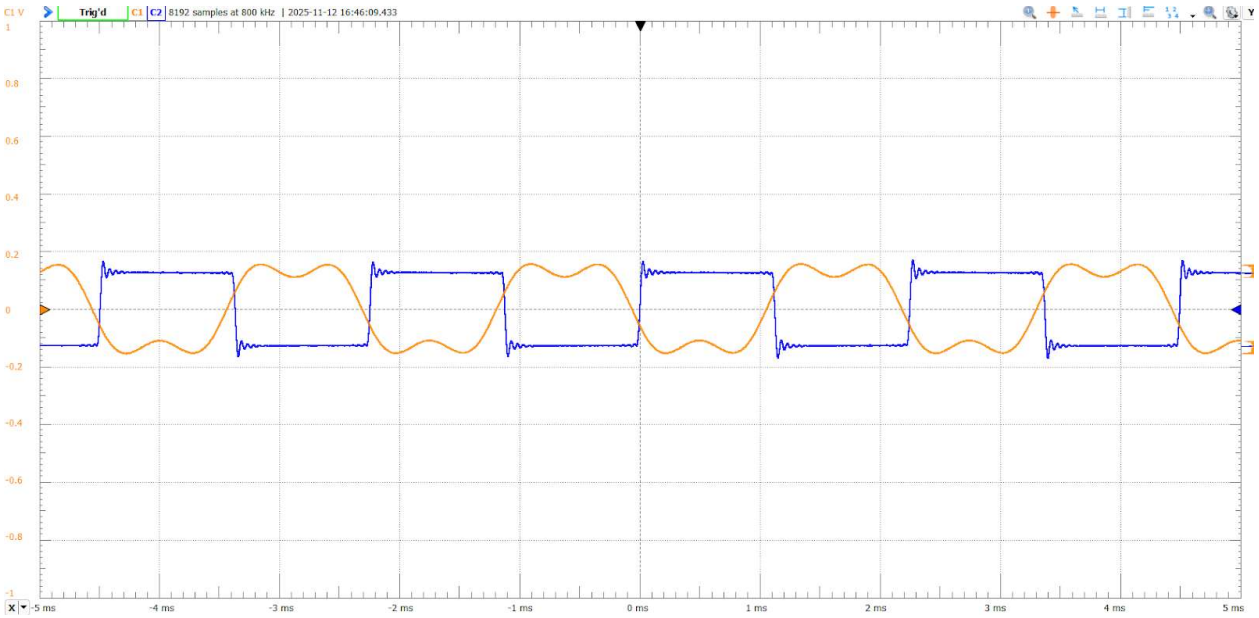


Figure 20. 445 Hz square waves through the transposed filter. Left (orange) is passed through the filter, right (blue) is not

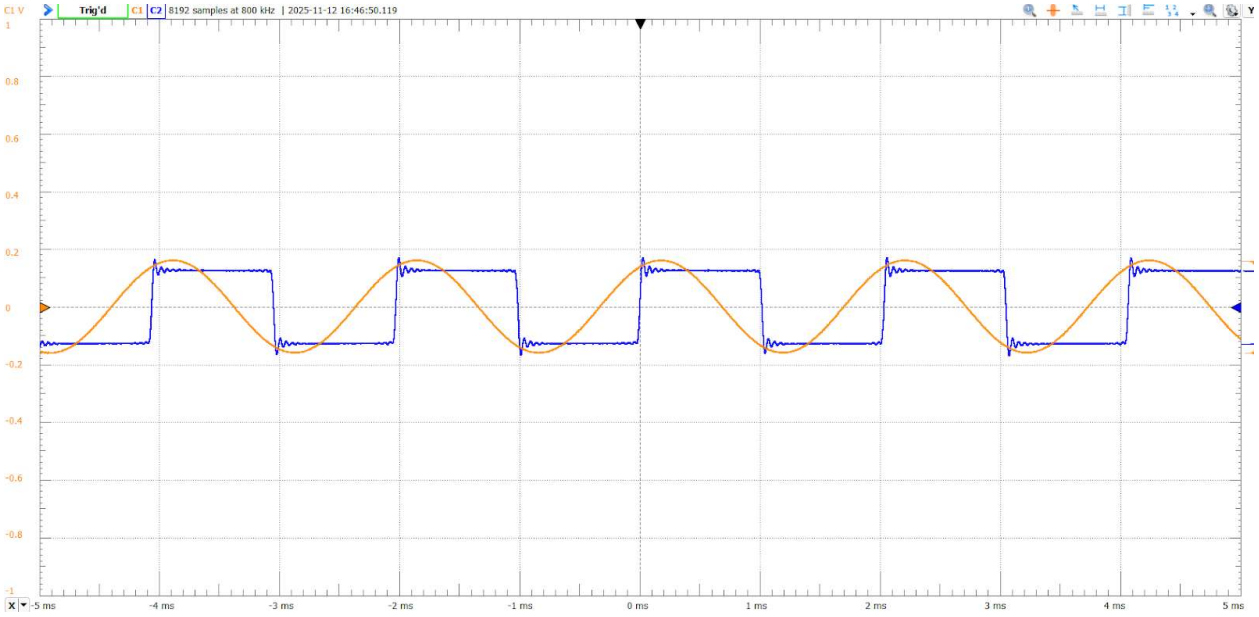


Figure 21. 492 Hz square waves through the transposed filter. Left (orange) is passed through the filter, right (blue) is not

```
S   ID  QUANT   RATE    WAIT    BUSY   W/Q   B/Q  ERR FORMAT          NAME
I   30      0      0   0.0us   0.0us  ???   ???    0                Dummy-Driver
S   31      0      0     ---     ---   ---   ---    0                Freewheel-Driver
S   52      0      0     ---     ---   ---   ---    0                Midi-Bridge
S   69      0      0     ---     ---   ---   ---    0                alsa_output.platform-fe00b840.mailbox.stereo-fallback
R   46   1024  48000   7.6ms   1.4us  0.36  0.00    0   S32LE 2 48000 alsa_input.platform-soc_sound.stereo-fallback
R   35      0      0  13.1us 146.0us  0.00  0.01    0   S32LE 2 48000  + alsa_output.platform-soc_sound.stereo-fallback
R   86      0      0  18.3us   7.4ms  0.00  0.35    0                  + simple
S   77      0      0     ---     ---   ---   ---    0                v4l2_input.platform-fe00b840.mailbox
S   79      0      0     ---     ---   ---   ---    0                v4l2_input.platform-fe00b840.mailbox
S   81      0      0     ---     ---   ---   ---    0                v4l2_input.platform-fe00b840.mailbox
S   83      0      0     ---     ---   ---   ---    0                v4l2_input.platform-fe00b840.mailbox
```

Figure 22. Performance of the transposed form FIR filter

Conclusion

We have shown how a lowpass filter can be created to separate DTMF tones. MATLAB filter designer can be used to create filters that meet exact requirements and then can be implemented. We have shown how an FIR filter can be constructed in both direct and transpose form, giving identical graphs but different performance.

After analyzing the performance of the three different forms of filters, direct, direct optimized, and transpose, it seems that the transpose form and the optimized version have comparable performance. However, the transpose is much simpler to code compared to the optimized direct form and therefore we believe it to be a better option for this case as the optimization difference is not enough to outweigh the inconvenience of complex code.

References

No external references were required for the completion of this lab.